OSC 源创会
年终盛典 2(

千人业务团队实施微服务的“血泪史”

李林锋：华为软件架构师
微博、微信：Nettying

引言 - 背景

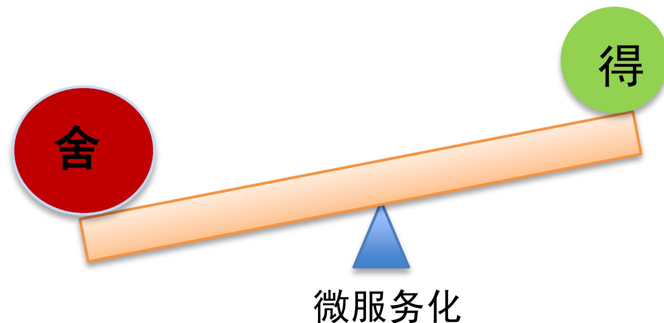
在过去的几年中，电信运营商积极进行数字化转型，技术体现主要就是云化（Docker容器化）和服务化（微服务化）。对于一些复杂电信系统(例如BSS)的服务化改造，是个极具挑战的任务...

实施前的准备工作 – 目标要清晰，处理好“舍与得”

教训：

- ✓ 微服务化目标不清晰：各业务模块处在不同技术阶段，有单体应用、RPC架构、SOA服务等，业务痛点不同。没有明确各业务的微服务化目标：提升开发效率、缩短新业务上线周期，还是单纯的追求技术先进性
- ✓ 没有处理好“舍与得”：微服务不是银弹，看着美好，但是要在大规模团队和电信复杂业务系统中实施，一定要处理好舍与得。例如分布式带来的时延增加、可靠性降低

经验总结：明确目标，敢于舍弃



- ✓ 性能下降，时延增加
- ✓ 环境搭建复杂
- ✓ 本地调试不方便
- ✓ 问题定位难度增大
- ✓ 不能随心所欲修改接口
- ✓ ...

- ✓ 业务更灵活的拆分
- ✓ 2 Pizza Team
- ✓ 持续构建和交付周期短
- ✓ 升级和部署更快，影响范围更小...
- ✓ 技术架构更开放
- ✓ ...

实施前的准备工作 – 微服务不仅仅是架构师的愿景

教训：

- ✓ 愿景美好，但落地变形：顶层架构设计仅仅存在于架构师的脑海和架构设计文档中，设计理念和关键架构属性底层的开发人员并不完全理解，架构落地时存在较大偏差
- ✓ 大兵团作战，各自为战：大规模业务团队拆分成很多小的微服务团队之后，需要互相协作和配合，在架构设计原则的理解和遵循上存在较大偏差，各自为战

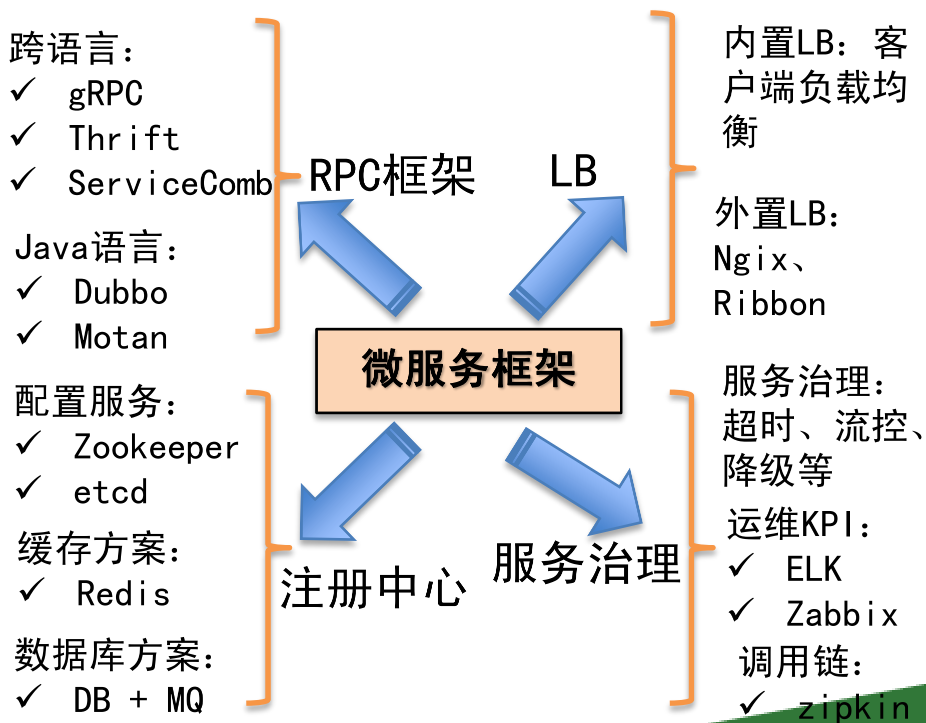
经验总结：统一认识，组织赋能

实施前的准备工作 – 技术选型要全面

教训：

- ✓ 过于关注运行态服务框架，而忽略其它质量属性：在选型时，过于关注微服务框架的性能指标、功能丰富性，以及对业务的侵入程度，而忽略了服务运维和治理
- ✓ 忽略微服务语言中立原则：业务模块多，场景复杂。大部分业务采用Java开发，但是对于计费、批价等性能要求苛刻的业务仍然需要继续使用C（C++）、GO等语言，服务框架不支持异构语言，跨语言调用存在问题

经验总结：分析全面，语言中立

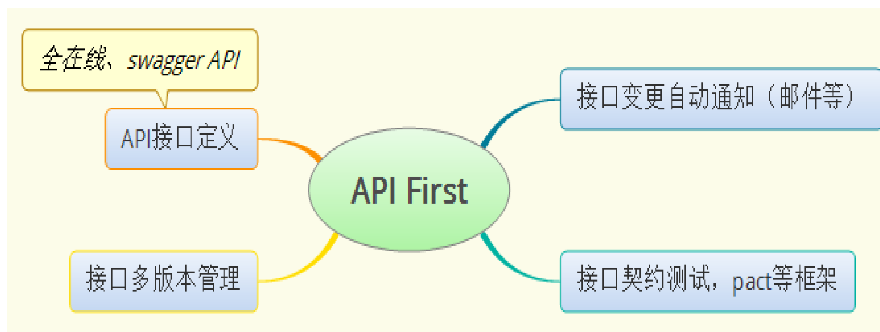


跨团队协作 – 接口依赖和进度协同问题多

教训：

- ✓ 无法及时提供接口API，影响其它团队开发进度：
  - a） 初期经验不足：服务提供者过分专注于微服务的内部实现，而不是优先设计微服务的API提供契约化的接口给消费者
  - b） 后期项目规则制约：限制迭代微服务接口变更次数。 提供者担心接口会经常的变更和重构，迟迟不提供接口契约
- ✓ 无有效的变更通知机制：微服务接口契约文档离线管理，变更之后无有效通知机制

经验总结：推行API First设计理念



- ✓ 消费者参与：推行API First的设计理念，让微服务提供者从消费者的角度，与消费者一起设计API，同时把API通过在线的方式发布和管控起来
- ✓ 全在线：基于swagger，可以通过YAML或者JSON的方式设计API、在线发布API

跨团队协作 – 接口兼容性问题

教训：

✓ 微服务版本管控意识不强，随意变更：随意删除、修改接口字段，导致其它团队CI构建失败，测试用例通过率低

✓ 完全基于管理手段：通过管理条例等约束接口变更，但是缺乏统一的技术保障手段。例如新增字段、删除字段，不同语言、不同团队怎么保证实施的一致性

经验总结：技术保障、管理协同

✓ 制定并严格执行《微服务前向兼容性规范》，避免发生不兼容修改或者私自修改不通知周边的情况

✓ 接口兼容性技术保障：例如Thrift的IDL，支持新增、修改和删除字段，字段定义位置无关性，码流支持乱序等。通过URL中携带V1/V2等主版本号，实现灰度路由

✓ 持续交付流水线的每日构建和契约化驱动测试，能够快速识别和发现不兼容

微服务实施 – 不是所有业务一刀切

教训：

- ✓ 研发阶段：为了技术架构的统一，不考虑业务之间的差异性，不同业务的痛点差异，一刀切的全部进行微服务化改造
- ✓ 上线阶段：一次性全量上线，所有微服务化改造的业务同时进行升级，没有观察期和缓冲期

经验总结：循序渐进、稳扎稳打

- ✓ 先外围，后中心：先找一些相对成熟，非核心模块的业务做微服务改造，在这个过程中不断积累经验，为后续核心模块的微服务化做准备
- ✓ 麻雀虽小五脏俱全：在微服务化早期实践中，除了开发工具和运行框架，需要把微服务持续交付流水线、微服务治理框架、调用链分析等配套设施全部构建起来
- ✓ 逐步上线和上量：上线时，可以采用灰度路由等方式，逐步把流量切换到新的微服务系统中来

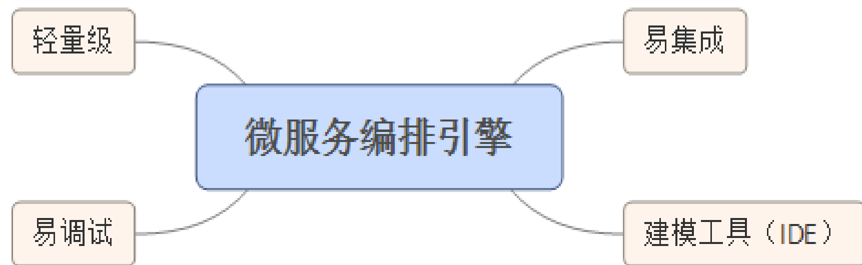
不可忽视的技术细节 － 微服务编排

教训：

- ✓ 编排框架混乱：硬编码/BPMN流程引擎、脚本（Groovy、JS）编排引擎等，缺乏统一的轻量级微服务编排引擎（PVM）
- ✓ 微服务编排层：顶层缺乏统一的设计和策略，导致各业务模块实现五花八门。有的在后台能力中心层、有的在中台，有的在API Gateway，有的在客户端

经验总结：轻量级微服务编排层

- ✓ 按需选择轻量级的微服务编排引擎（PVM），传统的BPMN在微服务时代模型较重、依赖较多
- ✓ 微服务的编排适合单独独立出来（中台），保障后台微服务的原子性、稳定性，同时又能够满足前台、移动端各种个性化需求



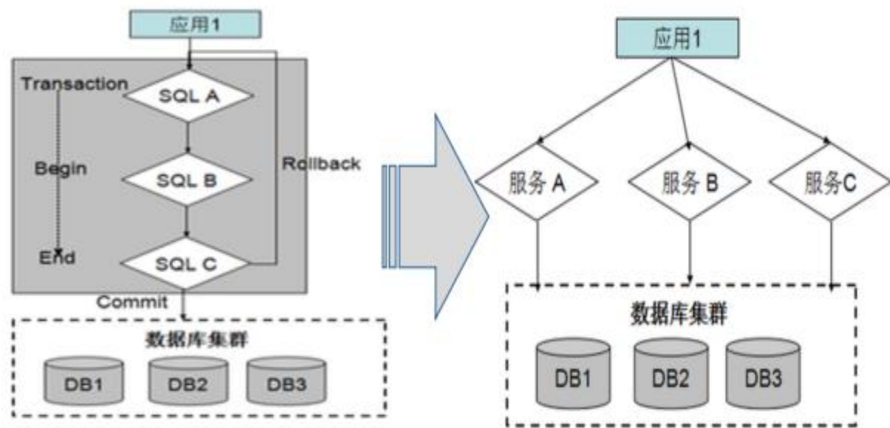
不可忽视的技术细节 – 分布式事务

教训：

✓ 分布式事务选型：业务补偿、事务强一致性（TCC框架）或者基于消息的最终一致性方案，平台和业务之间、不同业务团队之间没有达成一致性方案，影响事务一致性

✓ 习惯思维，强一致性事务占比过高：单体应用都是本地事务，强一致性容易保障。服务化之后，没有基于业务场景深入分析，习惯性的采用强一致性方案，业务成本很高

经验总结：以用户体验为本，兼顾时效性与成本



✓ 最终一致性，采用基于消息中间件的事务方案

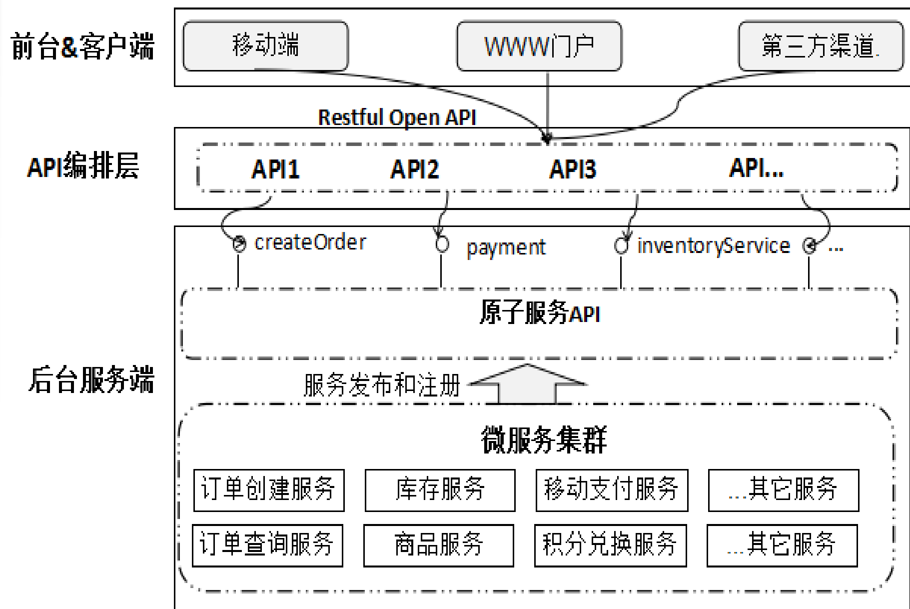
✓ 强一致性，TCC方案

不可忽视的技术细节 – API开放和集成

教训：

- ✓ 内部的实现细节开放给前端/第三方：包括但不限于特定语言的实现细节，例如异常类、继承和重载、抽象接口等。
- ✓ 为了后端数据结构重用，开放冗余的字段：有时候为了重用内部的数据结构，把整个对象都开放出去，事实上消费者只需要使用其中的几个字段，导致接口易用性变差
- ✓ 缺乏统一的 API Gateway：没有统一的API入口和精准的流量管控手段

经验总结：基于API Gateway，提升开放API的易用性



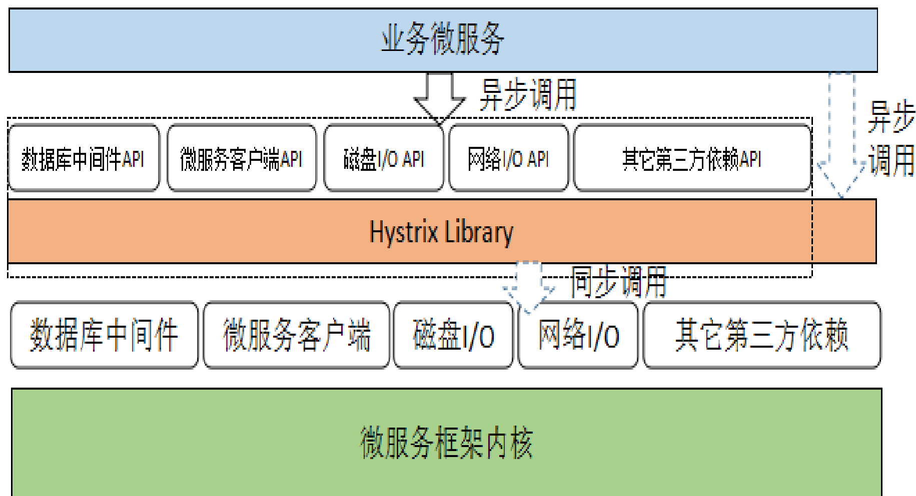
不可忽视的技术细节 – 熔断和降级

教训：

- ✓ 闭门造车，技术五花八门：没有构建统一的微服务可靠性框架、故障注入框架等，不同的业务团队各自为战。经验丰富的团队，可靠性做的较好，技能较差的团队，缺乏有效的可靠性保障
- ✓ 重复研发，资源浪费：业务场景不同，可靠性诉求大同小异。异步数据库访问、异步I/O操作、微服务故障隔离等能力，被不同团队重复构建

经验总结：集成业界成熟技术，统一构建可靠性和故障注入框架

- ✓ 对第三方依赖进行分类、分组管理，根据依赖的特点设置熔断策略、优雅降级策略、超时策略等，以实现差异化的处理



不可忽视的技术细节 – 微服务是否必须独立部署

教训：

- ✓ 机械照搬微服务原则，所有微服务都独立部署：上千个微服务，上百个节点集群组网，微服务进程数膨胀到数十万，周边配套设施无法承受（例如网管OSS）
- ✓ 为了简化部署和管理，微服务全部合设在同一个进程：丧失升级灵活性、无法按需伸缩、故障隔离差、异构语言无法合设在一起等缺点

经验总结：按照微服务的拆分粒度、微服务规模、以及运维团队能够接受的维护成本来决定微服务的部署策略

- ✓ 可以进程内合设的微服务

a） 性能、时延要求苛刻，需要本地短路的微服务可以合设

b） 业务强相关的一组微服务，为了便于统一管理

c） 暂时不支持分布式事务，需要本地事务保障强一致性的相关微服务（非长久之计）

不可忽视的技术细节 – 告警泛滥，坐卧不安

教训：

✓ 告警漫天飞，心惊肉跳：微服务化之后，本地的方法调用变成了远程的微服务调用，一个业务流程的多个微服务会有多个开发者负责。告警也由本地集中式演变成了分散的分布式告警，如果仍然沿用之前的告警方式，就会发现告警满天飞，而且大部分告警都是重复无效的告警

经验总结：动态构造告警树，自动抑制重复告警

✓ 原理

结合分布式调用链跟踪功能，在运行态将微服务调用关系动态刻画出来，然后构造告警树，每次叶子节点发生异常需要告警时，需要沿着树干层层下钻，对告警进行关联，寻找告警的Root节点，如果能够匹配上，则说明已经由根节点做了告警，叶子节点放弃告警，防止重复告警

平台和业务关系 – 并非甲方和乙方

教训：

- ✓ 以甲乙方来处理微服务平台和业务的关系：大规模业务团队，服务化需求五花八门，平台方需要识别高优先级、通用的需求，大部分个性化的需求需要开放给业务团队实现，或者规划到后续版本。如果业务方只从本团队利益出发，不顾全大局，很容易压垮平台，最终交付质量无法保障

经验总结：敢于和善于拒绝不合理需求

平台方：

- ✓ 合理沟通，据理力争
- ✓ 深刻理解业务场景，抓主要矛盾和矛盾的主要方面
- ✓ 敢于承担责任，敢于对不合理需求说“不”

业务方：

- ✓ 不断提升微服务实践经验，能够分辨哪些是平台需求，哪些是业务需求
- ✓ 与平台建立互信、合作共赢的新型关系

Q&A 环节

谢谢大家